|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| 1.12 Never,unknown and nullable type | {  // **nullable types / unknown types**   const searchName = (value: string | null) => {      if (value) { console.log("Searching"); } else { console.log("There is nothing to search"); }    };    searchName(null);  **// unknown  typeof**    const getSpeedInMeterPerSecond = (value: unknown) => {      if (typeof value === "number") {        const convertedSpeed = (value \* 1000) / 3600;        console.log(`The speed is ${convertedSpeed} ms^-1`);      } else if (typeof value === "string") {        const [result, unit] = value.split(" ");        const convertedSpeed = (parseFloat(result) \* 1000) / 3600;        console.log(`The speed is ${convertedSpeed} ms^-1`);      } else {        console.log("wrong input");      }    };    getSpeedInMeterPerSecond(null);  **//never**    const throwError = (msg: string): never => { throw new Error(msg); };    throwError("mushkil se error hogaya");    //  } |
|  |  |
| 2-2 Interface, type vs interface | { // **interface**    type User1 = { name: string; age: number; };    interface User2 { name: string; age: number; }    type UserWithRole1 = User1 & { role: string };    interface UserWithRole2 extends User1 { role: string; }    const user1: UserWithRole2 = { name: "Persian", age: 100, role: "manager", };    type rollNumber = number;    // js --> object , array-> object function -> object    type Roll1 = number[];    interface Roll2 { [index : number ] : number }    const rollNumber1: Roll2 = [1,2,3]                                0 1 2 --> number type    type Add1 = ( num1: number, num2:number )=> number    interface Add2 { (num1: number, num2:number) : number }    const add: Add2 = (num1, num2 )=> num1 + num2      //  } |
|  |  |
| 2-3 Introduction to generics | {  // **generic type**    type GenericArray<T> = Array<T>;    const rollNumbers1: number[] = [3, 6, 8];    const rollNumbers2: Array<number> = [3, 6, 8];    const rollNumbers3: GenericArray<number> = [3, 6, 8];    const mentors1: string[] = ["Mr. X", "Mr. Y", "Mr. Z"];    const mentors2: Array<string> = ["Mr. X", "Mr. Y", "Mr. Z"];    const mentors3: GenericArray<string> = ["Mr. X", "Mr. Y", "Mr. Z"];    const boolArray1: boolean[] = [true, false, true];    const boolArray2: Array<boolean> = [true, false, true];    const boolArray3: GenericArray<boolean> = [true, false, true];    interface User { name: string; age: number; }  const user1: GenericArray<{name: string; age: number;}> =[{ name: "Rony", age: 100,}, { name: "Rana", age: 110,}, ];  const user2: GenericArray<User> = [ { name: "Mezba", age: 100 }, { name: "Jhankar Mahbub", age: 110 }, ];    const add = (x: number, y: number) => x + y;    add(30, 20);    //generic tuple    type GenericTuple<X, Y> = [X, Y];    const manush: GenericTuple<string, string> = ["Mr. X", "Mr. Y"];    const UserWithID: GenericTuple<number, { name: string; email: string }> = [      1234, { name: "persian", email: "a@gmail.com" },  ];  } |
|  |  |
| 2-4 Generic with Interface | **{ // interface - generic**  interface Developer<T, X = null> {  name: string;  computer: { brand: string; model: string; releaseYear: number; };  smartWatch: T;  bike?: X;  }  type EmilabWatch = { brand: string; model: string; display: string; };  const poorDeveloper1: Developer< { brand: string; model: string; display: string; } > = {  name: "Person",  computer: { brand: "Asus", model: "X-kw66", releaseYear: 2013, },  smartWatch: { brand: "Emilab", model: "kw66", display: "OLED" },  };  const poorDeveloper2: Developer<EmilabWatch> = {  name: "Person",  computer: { brand: "Asus", model: "X-kw66", releaseYear: 2013, },  smartWatch: { brand: "Emilab", model: "kw66", display: "OLED" },  };  const richDevelper: Developer<{ brand: string; model: string; }> = {  name: "person",  computer: { brand: "HP", model: "x-25ur", releaseYear: 2018, },  smartWatch: { brand: "Apple Watch", model: "something", },  };  //} |
|  |  |
| 2-5 Function with generics | {**// function with generics**    const createArray = (param: string): string[] => { return [param]; };    const createArrayWithGeneric = <T>(param: T): T[] => { return [param]; };    const res1 = createArray("Bangladesh");    const resGeneric = createArrayWithGeneric<string>("Bangladesh");    const resGenericObj = createArrayWithGeneric<{ id: number; name: string }>(  { id: 222, name: "Mr. Rahman", }  );    type User = { id: number; name: string };    const resGenericObj1 = createArrayWithGeneric<User>( { id: 222, name: "Mr. Pashan", } );    const createArrayWithTuple = <T, Q>(param1: T, param2: Q): [T, Q] => {      return [param1, param2];    };    const res10 = createArrayWithTuple<string, number>("Bangladesh", 222);    const res11 = createArrayWithTuple<string, { name: string }>("Bangladesh", { name: "Asia", });    const addCourseToStudent = <T>(student: T) => {      const course = "Next Level Web Development";      return { ...student, course, };    };    const student1 = addCourseToStudent( { name: "Mr X", email: "x@gmail.com", devType: "NLWD", } );    const student2 = addCourseToStudent( { name: "Mr Y", email: "y@gmail.com", hasWatch: "Apple Watch", } );  // } |
|  |  |
| 2-6 Constraints in typescript | {  **// constraints**    const addCourseToStudent = <T extends { id: number; name: string; email: string } >( student: T ) => {      const course = "Next Level Web Development";      return { ...student, course, };    };    const student3 = addCourseToStudent( { id: 44, name: "Mr. Z", email: "z@gmail.com", emnis: "emni", } );    const student1 = addCourseToStudent<{ id: number; name: string; email: string; devType: string; }>(  { id: 222, name: "Mr X", email: "x@gmail.com", devType: "NLWD", }  );  const student2 = addCourseToStudent({ id: 333, name: "Mr Y", email: "y@gmail.com", hasWatch: "Apple Watch", });    // } |
|  |  |
| 2-7 Constraint using key of | {**// generic constraint with keyof operator**    type Vehicle = { bike: string; car: string; ship: string; };    type Owner = "bike" | "car" | "ship"; // manually    type Owner2 = keyof Vehicle;    const getPropertyValue = < X, Y extends keyof X >( obj: X, key: Y ) => { return obj[key]; };    const user = { name: "Mr. persian", age: 26, address: "ctg", };    const car = { model: "Toyota 100", year: 200,   };    const result1 = getPropertyValue( car, "model");    // obj[key]   26    // } |
|  |  |
| 2-8 Asynchronous typescript | // promise    type Todo = { id: number; userId: number; title: string; completed: boolean; };    const getTodo = async (): Promise<Todo> => {      const response = await fetch("https://jsonplaceholder.typicode.com/todos/1");      const data = await response.json();      return data;      // console.log(data);    };    getTodo();    type Something = { something: string };    // simulate    const createPromise = (): Promise<Something> => {      return new Promise<Something>((resolve, reject) => {        const data: Something = { something: "something" };        if (data) { resolve(data); } else { reject("failed to load data"); }      });    };    // calling create promise function    const showData = async (): Promise<Something> => {      const data: Something = await createPromise();      return data;      // console.log(data);    };    showData(); |
|  |  |
| 2-9 Conditional types | { **//conditional type**    type a1 = number;    type b1 = string;    type x = a1 extends null ? true : false; // conditional type    type y = a1 extends null ? true : b1 extends undefined ? undefined : any;    type Sheikh = { bike: string; car: string; ship: string; plane: string; };    //keyof Sheikh   "bike" | "car" | "ship"    // car ase kina / bike ase kina / ship kina / tractor ase kina    // type CheckVehicle<T> = T extends "bike" | "car" | "ship" | "plane" ? true : false;    type CheckVehicle<T> = T extends keyof Sheikh ? true : false;    type HasPlane = CheckVehicle<"plane">;    // } |
|  |  |
| 2-10 Mapped types | {   **// mapped types**    const arrOfNumbers: number[] = [1, 4, 5];    // const arrOfStrings : string[] = ['1','4','5']    const arrOfStrings: string[] = arrOfNumbers.map( (number) => number.toString() );    console.log(arrOfStrings);    type AreaNumber = { height: number; width: number; };    type Height = AreaNumber["height"]; // look up type    /\* type AreaString = { height: string; width: string } \*/    /\* type AreaString = { [key in "height" | "width"]: string; }; \*/    // keyof AreaNumber => "height"|"width"    /\* type AreaString = { [key in keyof AreaNumber ]: string; }; \*/    // T => {height:string;width:number}    // key => T["width"]    type AreaString<T> = { [key in keyof T]: T[key]; };    const area1: AreaString<{ height: string; width: number }> = { height: "100", width: 50, };    // } |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |